U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Faculdade de Engenharia
Universidade do Porto

Distributed and Partitioned Key-Value Store

Parallel and Distributed Computing, Project 2

Group 6
André Moreira up201904721@edu.fe.up.pt
Bruno Mendes up201906166@edu.fe.up.pt
Nuno Alves up201908250@edu.fe.up.pt

Friday 3rd June, 2022

Contents

Summary

In the age of always available digital information, it is imperative that service providers rely on a distributed infrastructure capable of handling loads of requests, while resisting to the ongoing environmental challenges, such as network or device failures.

This paper focuses on an implementation of a distributed, partitioned key-value store, with a design resembling the likes of *Amazon DynamoDB*. The consistency and availability of the service depends on mechanisms such as replication and fault tolerance between the service's cluster nodes, which are thoroughly described in the respective sections.

Message Format

2.1 Generic message format

Every message, between two nodes or between a client and a node, is composed of:

1. The header, composed of:

   Message type followed by `0x0D 0x0A`.

   Body size in bytes followed by `0x0D 0x0A`.

   Fields in a key-value fashion, each in a line containing the key followed by `0x20`, the value and `0x0D 0x0A`. This allows for string splitting on the processing side.

2. `0x0D 0x0A` used to separate headers from the body.
3. The body of the message, whose length in bytes is the same as the number on the second line of the message. This usually contains byte values read from one or multiple files.

2.2 Message specification

If a message extends another message, then it inherits the same headers. If a message is abstract, denoted by `(ABSTRACT)`, it is not instantiated and is used only to simplify message specification.

STATUS MESSAGE (ABSTRACT)

1. **Description:** Message that contains a status code in header
2. **Headers:**
   - **status code:** the status code representing success of an operation

KEY MESSAGE (ABSTRACT) extends STATUS MESSAGE

1. **Description:** Message that contains a key in header
2. **Headers:**
   - **key:** key of the pair

PUT extends KEY MESSAGE

1. **Description:** Message client sends to put a key-value pair on the cluster

2. **Body:** Value content in bytes

PUT RELAY

1. **Description:** Message sent between nodes to put/replicate pairs
2. **Headers:**
   - **numValues:** number of key-value pairs in message
   - **transference:** false if putting is forced
3. **Body:** The first **numValues** lines each contain two values separated by a space, on the left side are the keys and on the right side are the size of the respective values. After these first lines are the value's contents, one after the other, in order with the first lines, i.e. the first line corresponds to the first value read.

PUT REPLY extends KEY MESSAGE

1. **Description:** Message sent between node and client to acknowledge a put operation

PUT RELAY REPLY extends KEY MESSAGE

1. **Description:** Message sent between nodes to acknowledge one or more put operations
2. **Headers:**
   - **hashes:** successfully inserted hashes

GET extends KEY MESSAGE

1. **Description:** Message client sends to retrieve a value from the cluster with a key

GET RELAY extends KEY MESSAGE

1. **Description:** Message sent between nodes to retrieve a value from replication nodes

GET REPLY extends KEY MESSAGE

1. **Description:** Message the node sends to the client or to another node to send a value from the store
2. **Body:** Value content in bytes

DELETE

1. **Description:** Message client sends to delete a value from the cluster with a key

DELETE REPLY extends KEY MESSAGE

1. **Description:** Message the node sends to the client informing the status of a DELETE operation

DELETE RELAY

1. **Description:** Message a node sends to delete a value from another node with a key

ELECTION

1. **Description:** Message sent between nodes to define a leader
2. **Headers:**
   - **origin:** node's id of the origin of the message, node that wants to be leader
   - **membershipLog:** node's membership log

LEADER

1. **Description:** Message sent between nodes to inform the cluster nodes who the new leader of the cluster is
2. **Headers:**
   - **leaderNode:** id of the leader node

JOIN

1. **Description:** Message a node sends to signal intention to join or leave the cluster
2. **Headers:**
   - **nodeId:** the node id
   - **counter:** the node membership counter
   - **connectionPort:** the node port open for membership responses; unset on leave
   - **port:** the joining node port open for future client requests; unset on leave

MEMBERSHIP

1. **Description:** Membership message with sender cluster view information
2. **Headers:**
   - **nodes:** the list of cluster nodes, in the format *<nodeId0>/<port0>,<nodeId1>/<port1>...*
   - **nodeId:** the sender node id
3. **Body:** Sender membership log file bytes

The relay messages are an abstraction to allow for easier differentiation between operations on a node directly requested by the client or dispatched by another node, as explained in section 4.2.

Membership Service

The membership service of a node is aware of the state of the cluster and performs actions on its storage service. Its internal state, such as the view of the cluster nodes or the event log, is kept in files for easier administrator maintenance and for recovery purposes if the device crashes.

3.1 RMI interface

The definition of the remote interface can be seen in `server/MembershipRMI.java` and is implemented by `server/MembershipService.java`.

3.2 Joining the cluster

When a node is started its state is set to `InitNodeState` and starts accepting TCP connections on the specified IP and port on a new thread implemented by `server/tasks/MessageReceiverTask.java` and RMI calls. In this state the node responds to `leave` RMI calls with `ALREADY_LEFT` and to `PUT`, `GET`, and `DELETE` messages with an appropriate `PUT_REPLY`, `GET_REPLY`, or `DELETE_REPLY` message with an error code `NODE_NOT_JOINED`, as seen in `server/state/InitNodeState.java`.

When it receives the `join` RMI call, the joining process begins. First, the state is changed to `JoiningNodeState` which responds to `join` RMI calls with `JOIN_IN_PROGRESS` and processes `PUT_RELAY` and `DELETE_RELAY` messages (whose semantics are explained in section 4.2). Then, the node creates a multicast channel to the multicast group, but does not join it, and creates a `ServerSocket` bound to an IP same as the node's ID and with a random available port. Before sending the `JOIN` message on the multicast channel, it starts accepting connections on the just created `ServerSocket` for receiving the initial 3 `MEMBERSHIP` messages from 3 different nodes on another thread implemented by `server/tasks/JoinInitTask.java`. To do this, the `JOIN` message has a header `connectionPort` that specifies the port of the newly created socket. This new thread waits for receiving a `MEMBERSHIP` message; if it does not receive one after a timeout then it re-sends the `JOIN` message, and it stops after receiving all 3 `MEMBERSHIP` messages or after sending `2` `JOIN` messages, not counting the very first, closing the socket.

Nodes in the cluster, upon receiving the `JOIN` message, if the new node is found to be responsible for their keys send them to the joining node with `PUT_RELAY` and `DELETE_RELAY` messages, as described in sections 4.2 and 5.1. After, they wait a random amount of time before sending a `MEMBERSHIP` message to the specified port, if they have not sent one before for the same `JOIN` message or if the cluster view has changed (using a custom structure in `utils/SentMemberships`), as seen in `server/state/JoinedNodeState.java` in the `processJoinMessage` function. If no errors occur, the joining node sets its state to `JoinedNodeState`, writes the updated membership counter to a file, and joins the multicast group and listens for datagrams, starting a thread implemented by `communication/MulticastHandler.java`.

3.3 Leaving the cluster

If a node in the `JoinedNodeState` receives an RMI `leave` call it starts the leaving process. First, it sets its state to `InitNodeState` so as to not process any more messages as a member of the cluster. It also shuts down the thread pools of `MessageReceiverTask` and `MulticastHandler` and awaits for their termination, allowing for past messages to be processed and so that no messages interfere with the leaving process. The thread pool for `MessageReceiverTask` is restarted, so as to be able to send error messages to the Client. Afterwards, it sends the `JOIN` message with an incremented counter, leaves the multicast group, writes the counter, transfers its keys to the correct nodes using `relay` messages and clears its view of the cluster. This process can be seen in the `leave` function of the `server/state/JoinedNodeState.java` class.

Nodes in the cluster, upon receiving the `JOIN` message with odd membership counter, remove the node from the cluster view and update their membership log. The behaviour described is shown in `server/state/JoinedNodeState.java` in the `processLeaveMessage` function.

3.4 Merging logs and view

Upon reception of a membership message, either via join response or via the periodic leader message, a node accepts the events on nodes it does not know, adding them to its view if their counter is even, and on nodes which counter is bigger, i.e. more updated, than the one locally recorded, removing them from its view if their counter is odd, and updating the key distribution.

This simple algorithm keeps the cluster view updated across the cluster, allowing the fault healing measures to take place and is implemented in the `processMembership` function of the `server/state/CommonState.java` class.

3.5 Leader election

The cluster *leader* is the node responsible for multicasting its membership log to the cluster, each second. Since it should be the most updated one, an algorithm taking advantage of the ring implementation of the key-value store was implemented to keep the leader a capable one.

When a node joins the cluster, its leadership state is set to `false`, proceeding to start a leader election process. After the initial process, each node in the cluster, periodically, each 10 seconds, starts a new leader election process, trying to discover if itself is the leader. This is implemented in `server/tasks/ElectionTask.java`

As a node starts the election process, it will send an `ElectionMessage` to his successor, containing information about its membership log. Upon receiving an `ElectionMessage`, a node will verify which is the most updated node, either itself or the incoming one, comparing the logs. If both nodes are equally updated, the node id is used to disambiguate. This process is implemented in the `processElection` function of the `server/state/JoinedNodeState.java` class.

If the node attempting to be the leader is more updated, then the current node passes the message to its successor. If an election message reaches the original sender, verified by comparing the node id with the `origin` attribute in the message, it is the leader. In this moment, the node sets itself as the leader.

After finding the leader, the whole cluster needs to be informed so that the old leader is revoked. As soon as the new leader is setup, it sends a `LeaderMessage` to every node in the cluster informing, in the message attribute, who the new leader is. The processing of this message is implemented in the `processLeader` function of the `server/state/JoinedNodeState.java` class.

3.6 Fault tolerance

If, upon dispatching the election message to its successor, a node finds it is unavailable, it is removed from its cluster view, and a `leave` event is generated for the down node. Key transferals, to keep the replication factor, described on section 5.1, are also performed. This is implemented in the `sendToNextAvailableNode` function of the `server/MembershipService.java` class. The node which finds the new failure event is the leader in a few seconds, and all nodes are informed of it.

When a down node wakes up from its failure (e.g. the network is back), it receives membership logs with its own counter set to a odd number, one more than the one it thinks it has. It thus acknowledges that it has crashed, increments its membership counter by two (resulting in one more than the one it received, so that peer nodes recognize its health), and starts a join process (section 3.2), so that it receives the keys it should have by the ring configuration, deletes keys that were already deleted without it knowing and transfers the keys it should not have. This is implemented in the `processMembership` function of the `server/state/CommonState.java` class.

During the joining or leaving process, the membership counter is only written to file whenever the node has really joined or left the cluster, so that, if an error or a crash occurs, the changes to the counter are rolled back.

Storage Service

The service cluster is composed of a set of nodes, disposed in a ring fashion, ordered by their IDs, in a consistent hashing manner. This allows for quicker and lesser key transferals when a node joins or leaves the cluster.

4.1 Determining the responsible node

The nodes responsible for a key are three nodes that immediately follow it in the ring. They are stored in a `TreeMap` (see `utils/ClusterMap.java`), by their IDs, which in Java is implemented using a binary search tree, speeding up the lookup operations.

This topic is further discussed in section 5.1 on the *Replication* chapter.

4.2 Transferring keys between nodes

Messages exchanged between the cluster itself carry relevant information for the receiver to process it. While a regular get, put or delete message result in the node calculating the successors of the hash, the redirects to them will follow in the form of a `relay` message, which is a passive equivalent of the former, causing the receiver to just perform the action on their local storage, without further dispatches.

The `PUT RELAY` message is also special because it allows to send multiple key-value pairs on the same message, so as to not flood the node's message receiver threads with many `PUT RELAY` messages. This message type can only hold a total of 100 key-value pairs as a form of minimizing network transferral time (see `message/PutRelayMessage.java`).

The operations are acknowledged through reply messages, which contain a status code related to the exit of the operation. This serves not only to save time (for example, a successful get operation on the first key successor means that the two following nodes do not need to be requested), but also to inform the client if the operation fails on all nodes.

4.3 Fault tolerance

The careful, frequent election of an appropriate cluster leader should keep the cluster view updated. If, for some pressing phenomenon, a node cannot update its view with the more recently joined cluster nodes, and it does not acknowledge it has been down for it to join the cluster again and receive fresh cluster information, it may send a hash to the wrong node. This, however, is not a so relevant issue: after deletion, get requests will not be able to reach the said key unless via the faulty node which is convinced of the wrong responsible; the others will go through the three correct, responsible nodes, which will have tombstones and return error. Also, any join or leave event on the incorrectly dispatched node will put the key where it belongs.

Other fault healing measures were already described in 3.6.

Replication

5.1 Keeping the replication factor

In order to maintain information safe against nodes' failure, the system must have replicated information among other nodes.

When a client requests a put, get or delete operation to a node, the three responsible nodes for storing the value will receive a redirected message - relay message - from the node responsible for answering the request from the client. After processing each reply from the responsible nodes, the reply with the best status code is transmitted to the client (see function `processPut` in class `server/state/JoinedNodeState.java`).

In order to maintain the replication factor, as soon as the nodes find an unavailable node, the node is removed and the request is sent to the next successor, updating the responsible nodes (see functions `removeUnavailableNode` and `transferMyKeysToNodes` functions in `server/MembershipService.java`).

The actions taken on membership changes, signaled by a join message, a leave message or a membership message, also aid to keep the replication factor constant (see section 3) as they call the `transferMyKeysToNodes` function.

5.2 Tombstones

If a node is down and misses a delete action on a key it has, it may try to propagate that key to new nodes that it acknowledges on the first new membership after it revives. This would make the key available again for get requests when it is not supposed to be.

This issue is solved by marking deleted files as *tombstones*, an empty file, put on a different folder, with the same hash (see `delete` in `server/StorageService.java`).

This way, upon receiving a put message from another node, the addition of a tombstoned key should only be performed if it originated from a client request (the `isTransference` field of the relay message is false); otherwise, the key is being incorrectly propagated (see `processDelete` function in the `server/state/JoinedNodeState.java` class).

Delete requests for all locally tombstoned hashes are dispatched to joining nodes, to avoid phenomenons like the aforementioned (see `orderJoiningNodeToDeleteMyTombstones` in `server/MembershipService.java`).

Concurrency

6.1 Thread pools

Thread Pools are used to achieve multi-threading without needing to create a Thread every time a new job has to be processed. There are two thread pools, both with a fixed thread number of `Runtime.getRuntime().availableProcessors()/ 2`.

The first is defined and used in `server/tasks/MessageReceiverTask.java` which handles received TCP messages and queues a job for each received message in the thread pool queue. The second is in `communication/MulticastHandler.java` which handles received multicast messages and sends message processing jobs to the thread pool queue.

6.2 Thread tasks

Thread tasks are threads the node creates that are not associated with the two previously mentioned thread pools.

The `server/tasks/ElectionTask.java` and `server/tasks/MembershipTask.java` threads are created when the node is created and are activated using two scheduled thread pools with one thread each with a fixed rate of 10 seconds for the `ElectionTask` and 1 second for the `MembershipTask`. Their function is described in section 3.5.

The `server/tasks/MessageReceiverTask.java` thread is created at node creation and runs until the node is closed. It handles the thread pool described in section 6.1. The `server/tasks/JoinInitTask.java` thread is created when the node starts its joining process, so it can start listening for the `MEMBERSHIP` messages before sending the `JOIN` multicast message, as described in section 3.2. The `communication/MulticastHandler.java` thread is created when the node sets its state to `JoinedNodeState` and is described in section 6.1.

6.3 Race conditions and deadlocks

Special care was taken to avoid race conditions and deadlocks, while also maintaining a fast node response time.

Some collections are created using concurrent implementations. In `StorageService`, `hashLocks` is a `synchronizedMap`, and `tombstones` is a `synchronizedSet`. In `ClusterMap`, `clusterNodes` is a `synchronizedSortedMap`. In `MembershipLog`, `membershipLog` is a `synchronizedMap`. In `SentMemberships`, `sentMemberships` is a `ConcurrentHashMap`.

Functions that update a variable and write the result to a file are made synchronized to guarantee atomicity. Iteration over concurrent collections is also wrapped with a synchronized block using as a lock the collection being iterated.

The join and leave RMI calls are locked using a special mutex `joinLeaveLock` so that the node does not handle multiple join and leave calls at the same time.

Every hash a node has is associated to a mutex lock in the structure `hashLocks` in `StorageService`. Processing `GET`, `PUT`, `DELETE`, and their relay counterpart messages is locked between processing of the same messages on the same hash, so as to not have concurrency while writing and reading to files. The transferral of keys is also locked using the same mutexes for each key so that this process does not happen concurrently with the `PUT`, and `DELETE` messages.

Conclusion

Implementing a robust, distributed service is a complex engineering challenge. The need for high performance and concurrency brings synchronization problems, which may result in catastrophic, wrong dispatches of actions, leading to possibly affect user's data privacy or simply make it unavailable.

This implementation aims to avoid the aforementioned issues, by dynamically updating the cluster view and carefully handling race conditions. While this does not come without some impact on performance, it should be noted that there is still room for optimization, namely via the reduction of threads, through *asynchronous IO*, or the reduction on the number of messages, for example, asking a node if it is already holding a key before sending the whole key/value.